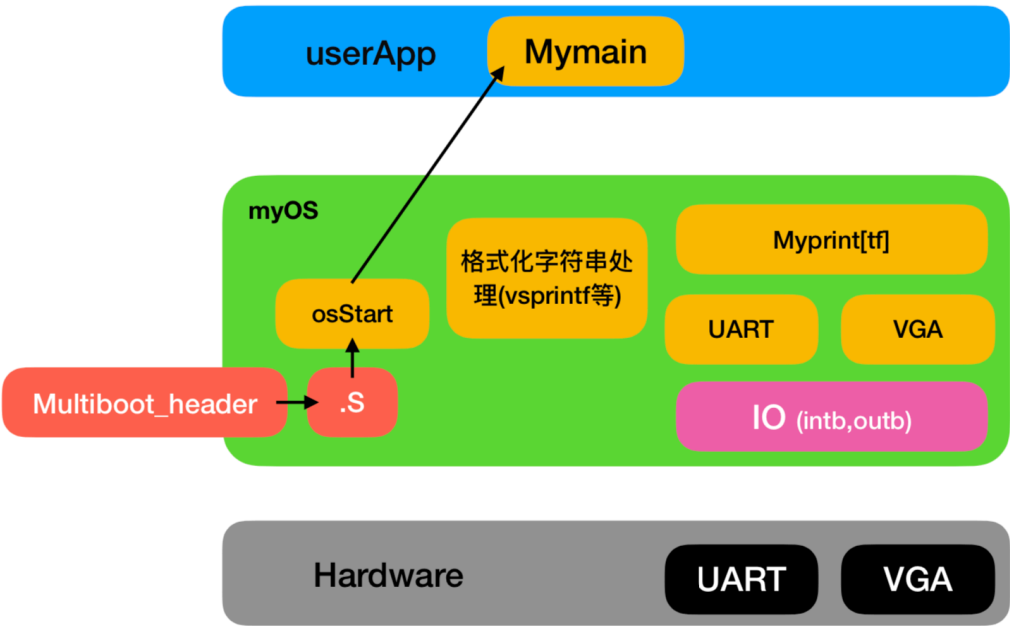


实验报告二：Multiboot2myMain

软件架构及说明

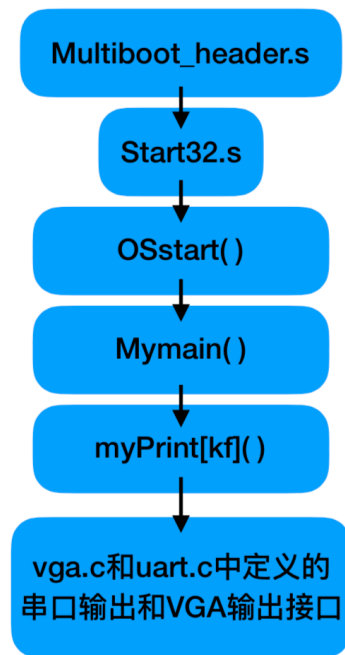
软件框图如下：



软件框图

说明：从Multiboot_header进入操作系统内核(myOS)，然后开始为进入C程序准备好上下文，初始化操作系统，再调用userApp入口myMain。在myMain中测试功能，包括通过调用Myprint[tf]函数实现VGA输出和串口的UART输出。

主流程及其实现



主流程图

主流程说明：multiboot_header.s调用_start入口，myOS中的Start32.s提供了_start()入口，做好第一次调用C语言入口前的准备，以osStart为myOS的第一个C入口，进入osStart后，先初始化操作系统，再调用与userApp之间的接口myMain，进入myMain后，调用函数myPrintf()、myPrintk()，这些是用户自定义的函数，调用了其他接口实现VGA和串口输出功能。

主要功能模块及其实现（含源代码说明）

模块一 IO：端口输入输出

源代码：io.c

```
/* IO operations */
unsigned char inb(unsigned short int port_from){
    //参考下面的outb函数，实现inb函数
    unsigned char _in_value;
    __asm__ __volatile__ ("inb %w1,%0"::"a"(_in_value):"Nd"(port_from));

    return _in_value;
}

void outb (unsigned short int port_to, unsigned char value){
    __asm__ __volatile__ ("outb %b0,%w1"::"a" (value),"Nd" (port_to));
}
```

代码说明：提供outb和inb函数实现端口输入输出，采用嵌入式汇编。inb()接收一个端口号作为参数，返回一个该端口接收到的值_in_value；outb()接收一个端口号和一个值value作为参数，将该值输出到指定端口。

__asm__ __volatile__ 的意义：asm表示后面的代码为内嵌汇编，volatile表示编译器不要优化代码,后面的指令保留原样。括号里面是汇编指令。

模块二 UART：串口输出

源代码：uart.c

```
//调用inb和outb函数，实现下面的uart的三个函数
extern unsigned char inb(unsigned short int port_from);
extern void outb (unsigned short int port_to, unsigned char value);

#define uart_base 0x3F8

void uart_put_char(unsigned char c){

    if(c == '\n'){           //特殊处理'\n'符
        outb(uart_base, '\r');
        outb(uart_base, '\n');
    }
    else{
        outb(uart_base, c);
    }

}

unsigned char uart_get_char(void){
    return inb(uart_base);
}

void uart_put_chars(char *str){
    char *ptr = str;
    char c;

    c = *ptr;
    while (c!='\0'){
        uart_put_char(c);    //逐个字符调用uart_put_char实现串口输出
        c = *(++ptr);
    }
}
```

代码说明：一般从uart_put_chars()开始接收待输出字符串。uart_put_chars()内部调用了uart_put_char()对字符串逐个字符输出。uart_put_char()以单个字符和预先定义好的端口地址0x3F8作为参数，并将参数传给outb()执行端口输出。

模块三 VGA：清屏、屏幕彩色输出（带滚屏）

源代码: vga.c

//本文件实现vga的相关功能, 清屏和屏幕输出, clear_screen和append2screen必须按照如下实现, 可以增加其他函数供clear_screen和append2screen调用

```
extern void disable_interrupt(void);
extern void enable_interrupt(void);

extern void outb (unsigned short int port_to, unsigned char value);
extern unsigned char inb(unsigned short int port_from);

int cursor_row, cursor_col;      //光标

void clear_screen(void) {
    int row, col;
    unsigned short *ptr = (unsigned short *)0xb8000;
    for(row = 0; row < 25; row++) {
        for (col = 0; col < 80; col++) {
            (*ptr++) = 0;
        }
    }
}

void put_char(char c, char color) {

    int i;
    unsigned char *ptr = (unsigned char *)0xb8000;
    unsigned int pos;

    if(c == '\n' || cursor_col == 80)
    {
        cursor_col = 0;
        cursor_row = (cursor_row + 1);
        if(cursor_row==25)      //即将执行滚屏操作
        {
            for(i = 0; i < 24*80*2; i++)
                ptr[i]=ptr[i+80*2];
            for(i = 24*80*2; i < 25*80*2; i++)
                ptr[i]=0;
            cursor_row=24;
        }
    }

    if(c!='\n')
    {
        ptr[(cursor_row * 80 + cursor_col) * 2] = c;      //写入字符值
        ptr[(cursor_row * 80 + cursor_col) * 2 + 1] = color;      //写入字符颜色
        cursor_col ++;
    }
}
```

```

    }

    ptr[(cursor_row * 80 + cursor_col) * 2 + 1] = 0x7; //设置光标颜色
    pos=cursor_row*80+cursor_col; //设置光标位置
    outb(0x3d4,14);
    outb(0x3d5,(pos>>8) & 0xff);
    outb(0x3d4,15);
    outb(0x3d5,pos & 0xff);
}

void append2screen(char *str,int color){
    char *ptr = str;
    char c;

    c = *ptr;
    while (c!='\0'){
        put_char(c, color); //逐个字符调用put_char()实现VGA输出
        c = *(++ptr);
    }
}

```

代码说明：从append2screen()开始接收待输出字符串和颜色格式参数，append2screen()内部调用了put_char()对字符串逐个字符输出。put_char()直接将字符值和显示属性信息写到VGA显存以实现VGA输出，通过调用outb()设置光标位置，通过将显存第二行开始的每一行数据向上移动(覆盖)一行实现滚屏。另外clear_screen()函数通过将显存数据全部清零实现清屏操作。

模块四 myPrint[kf]

源代码：myPrintk.c

```

//实现myPrint功能，需要调用到格式化输出的function (vsprintf)
#include <stdarg.h> //需要提供实现可变参数功能的函数和宏
#include "vsprintf.c" //需要引用移植的vsprintf()
extern void append2screen(char *str,int color);
extern void uart_put_chars(char *str);
extern int vsprintf(char *buf, const char *fmt, va_list args);

char kBuf[400]; //TODO: fix me
int myPrintk(int color,const char *format, ...){
    va_list ap;
    va_start(ap,format);
    vsprintf(kBuf,format,ap); //调用vsprintf处理格式化字符串，结果存储在kBuf[]中
    va_end(ap);
    uart_put_chars(kBuf); //调用串口输出
    append2screen(kBuf,color); //调用VGA输出
}

char uBuf[400]; //TODO: fix me
int myPrintf(int color,const char *format, ...){

```

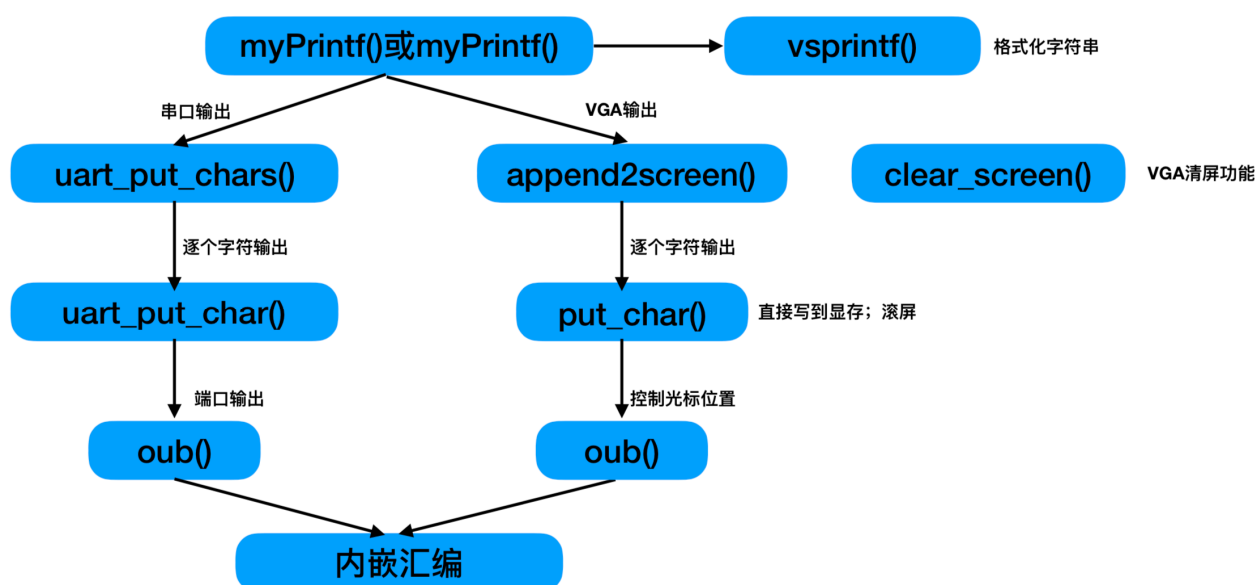
```

va_list ap;
va_start(ap, format);
    vsprintf(uBuf, format, ap);    //调用vsprintf处理格式化字符串，结果存储在uBuf[]中
    va_end(ap);
    uart_put_chars(uBuf);          //调用串口输出
    append2screen(uBuf, color);    //调用VGA输出
}

```

代码说明：myPrintf()内部调用了网上移植的vsprintf()处理格式化字符串，并将实际打印的字符存储在字符数组uBuf中，再把uBuf传给uart_put_chars()和append2screen()分别执行串口输出和VGA输出。myPrintk()与myPrintf()操作相同，仅用来存储字符串的数组不同。

主要功能模块流程图如下所示：



主要功能模块流程图

目录组织(编译前)：

```

.
├── Makefile
├── README_multiboot2myMain.txt
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── Makefile
│   ├── dev
│   │   ├── Makefile
│   │   ├── uart.c
│   │   └── vga.c
│   ├── i386
│   │   ├── Makefile
│   │   ├── io.c
│   │   └── io.h
│   ├── myOS.ld
│   ├── osStart.c
│   ├── printk
│   │   ├── Makefile
│   │   ├── myPrintk.c
│   │   └── vsprintf.c
│   └── start32.S
├── source2run.sh
├── userApp
│   ├── Makefile
│   └── main.c
└── 6 directories, 19 files

```

Makefile组织:

```

#注意替换SRC_RT, 其他内容不需要修改
SRC_RT=/home/lps3025/workspace/2_multiboot2myMain/
#$(shell pwd)

CROSS_COMPILE=
ASM_FLAGS= -m32 --pipe -Wall -fasm -g -O1 -fno-stack-protector
C_FLAGS = -m32 -fno-stack-protector -g

.PHONY: all
all: output/myOS.elf

MULTI_BOOT_HEADER=output/multibootheader/multibootHeader.o
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile

OS_OBJS      = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c

```

```
@mkdir -p $(dir $@)
@${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<

clean:
    rm -rf output
```

基本规则：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf
```

将各级子目录下makefile文件中定义好的.o文件作为依赖文件，按照myOS.ld规则生成目标文件myOS.elf，放在output文件夹下。

代码布局说明

链接描述文件myOS.ld如下：

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(start)

SECTIONS {
    . = 1M;
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

    . = ALIGN(16);
    .data : { *(.data*) }

    . = ALIGN(16);
    .bss :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
    . = ALIGN(512);
}
```


说明：首先定位到1M地址处。可执行文件的.text段从此处开始。开始存放输入文件(MultibootHeader.S)的.multiboot_header段[12字节]，往后对齐8字节后，再存放所有输入文件的.text段。往后对齐16字节后，接着存放可执行文件的.data段，包含的内容即为所有输入文件的.data段。往后对齐16字节后，接着存放可执行文件的.bss段，包括所有输入文件中一些未初始化的变量。bss段结束后依次向后对齐16字节和512字节。

编译过程说明

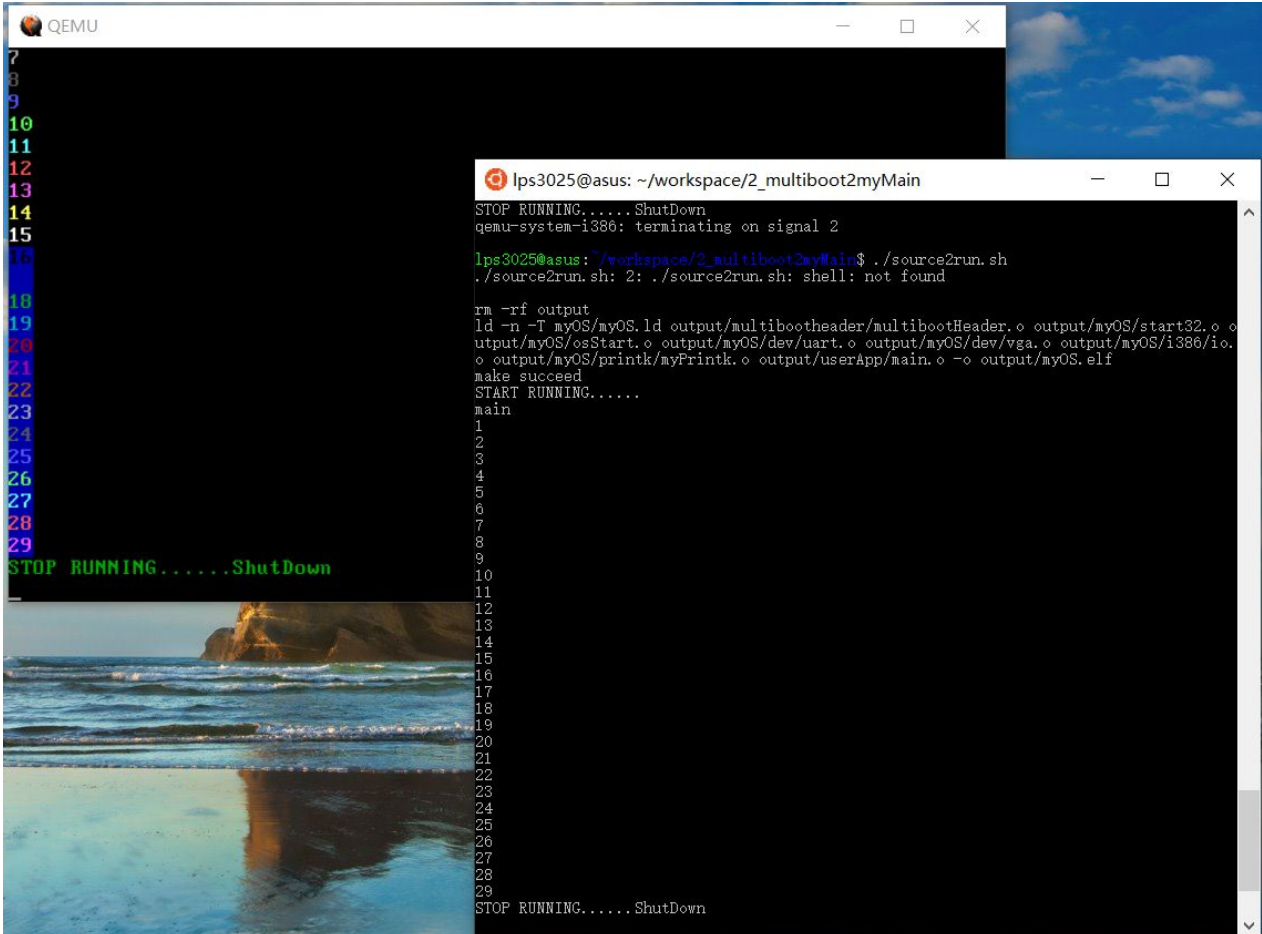
直接在终端运行./source2run.sh即可。具体过程是先make编译，编译成功后再执行命令

```
qemu-system-i386 -kernel output/myOS.elf -serial stdio
```

将生成的操作系统内核myOS.elf加载到qemu上运行。

运行和运行结果说明

运行结果如下：



```
QEMU
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
STOP RUNNING.....ShutDown

lps3025@asus: ~/workspace/2_multiboot2myMain
STOP RUNNING.....ShutDown
qemu-system-i386: terminating on signal 2

lps3025@asus: ~/workspace/2_multiboot2myMain$ ./source2run.sh
./source2run.sh: 2: ./source2run.sh: shell: not found

rm -rf output
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start32.o o
output/myOS/osStart.o output/myOS/dev/uart.o output/myOS/dev/vga.o output/myOS/i386/io.
o output/myOS/printk/myPrintk.o output/userApp/main.o -o output/myOS.elf
make succeed
START RUNNING.....
main
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
STOP RUNNING.....ShutDown
```

说明：终端上先打印出"Start RUNNING....."和"main"，接着循环输出数字1~29，最后打印出 "STOP RUNNING.....ShutDown"。SDL窗口上的输出内容相同，每一行字符颜色按指定颜色来，因添加了滚屏功能所以只能看到从数字7开始的行，最后一行有闪烁的光标。

遇到的问题和解决方案

第一次编译时出现报错：

```
fatal error bits/libc-header-start.h no such file or directory
#include<bits/libc-header-start.h>
```

错误出现在移植的vsprintf.c文件中 include<string.h>这一行代码。网上查询后得知是因为在64位机器上gcc缺少32位的库。

解决方案：执行sudo apt-get install gcc-multilib来安装32位的库。

后来vsprintf.c中又有报错：

```
undefined reference to 'strlen'
```

估计又是因为库中缺少了相关函数。

解决方案：自己写一个my_strlen()来替代strlen()的功能。

另外在实现VGA输出时，一直无法在光标位置显示出光标。测试发现光标颜色取决于当前字符的前景色。若显存中当前光标位置对应存储单元的值为全0则无法显示出光标。

解决方案：在光标位置补了一个color字节来指定光标颜色，值为0x7。